Software Engineering Fundamentals: MS.NET

Exception Handling

accenture

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

Technology Solutions

Objectives

- At the end of this module you should be able to:
  - Define an exception.
  - Explain how C# handles exceptions.
  - Explain the concept of common exception classes.
  - Demonstrate try – catch – finally statement.
  - Demonstrate how to propagate exceptions using the throw statement.
  - Identify user-defined exceptions.
  - Identify best practices for handling exceptions.

Agenda

- What is an Exception?
- Exception Handling in C#
- Common Exception Classes
- `try – catch – finally` statements
- Unhandled Exceptions
- Handled Exceptions
- Propagating Exceptions
- User-Defined Exceptions
- Best Practices for Handling Exceptions
- Key Points

What is an Exception?

- Exception is
  - An event during program execution that prevents the program from continuing normally.
  - An error condition that changes the normal flow of control in a program
  - A signal that some unexpected condition has occurred in the program

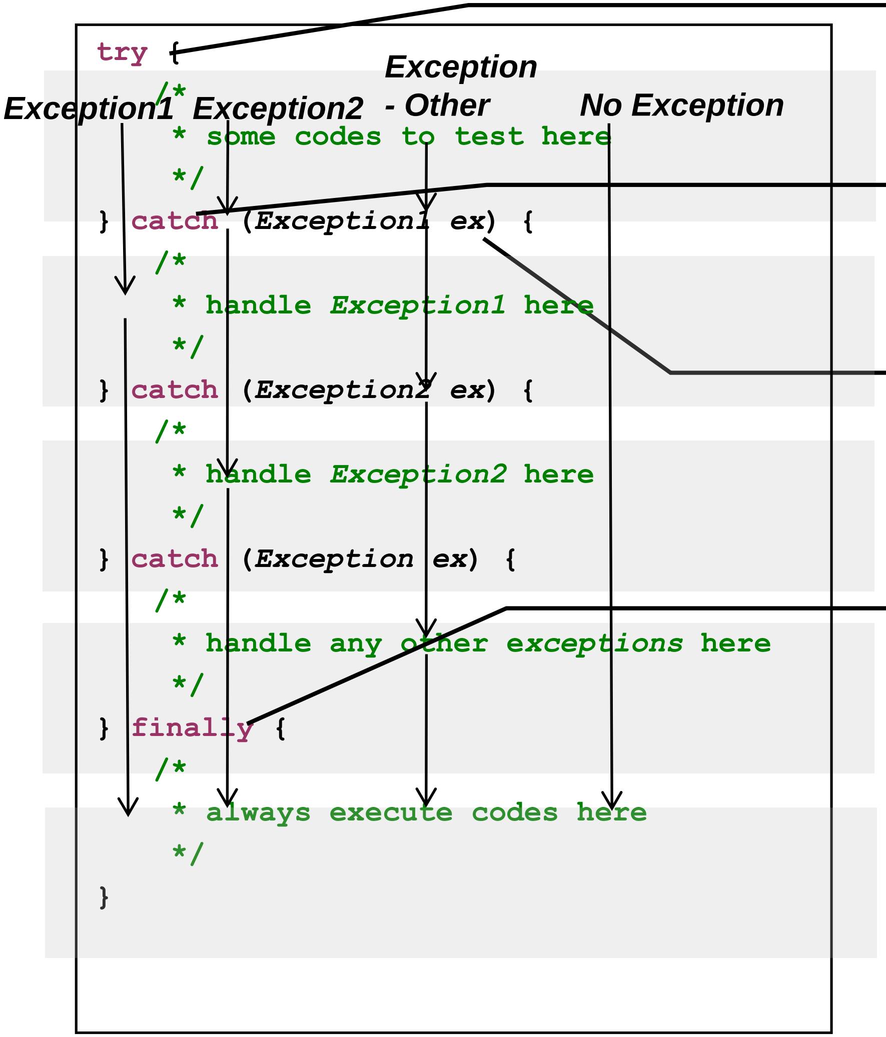
Exception Handling in C#

- Exceptions are handled by using a try statement in C#
- When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the run-time type of the exception.
  - First, the current method searches for a lexically enclosing try statement, and the associated catch clauses of the try statement are considered in order.
  - If that fails, the method that called the current method searches for a lexically enclosing try statement that encloses the point of the call to the current method. This search continues until a catch clause is found to handle the current exception.

Exception Handling in C# (cont.)

- If no matching **catch** clause is found, one of two things occurs:
  - If the search for a matching **catch** clause reaches a **static** constructor or **static** field initializer, then a *System.TypeInitializationException* is thrown at the point that triggered the invocation of the static constructor.
  - If the search for matching **catch** clause reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

try-catch-finally



```
try {
    /*
     * some codes to test here
     */
} catch (Exception1 ex) {
    /*
     * handle Exception1 here
     */
} catch (Exception2 ex) {
    /*
     * handle Exception2 here
     */
} catch (Exception ex) {
    /*
     * handle any other exceptions here
     */
} finally {
    /*
     * always execute codes here
     */
}
```

`try` block encloses the context where a possible exception can be thrown

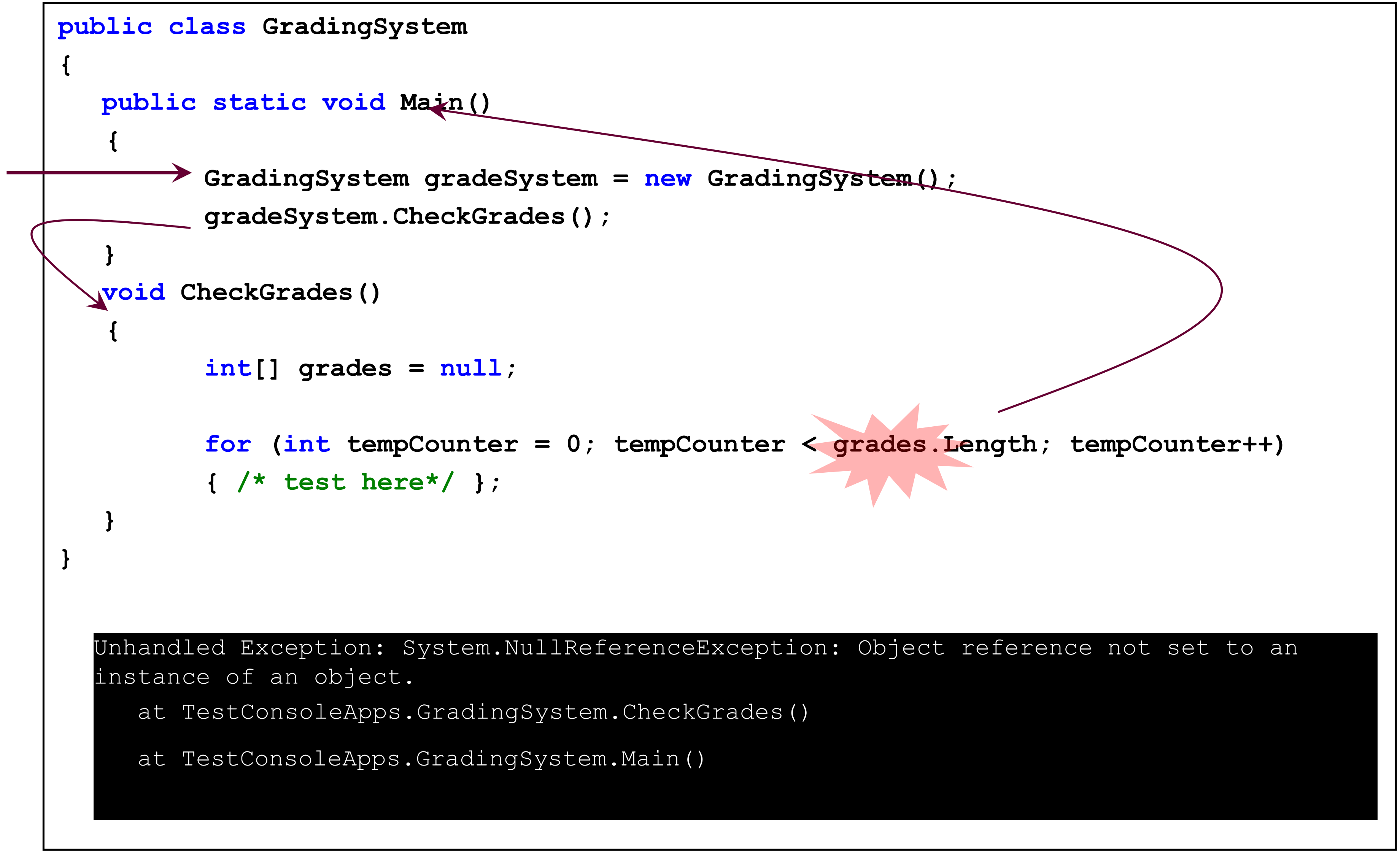
each `catch()` block is an exception handler and can appear several times

Exception1 should not shadow *Exception2* which in turn should not shadow *Exception* (based on the exception hierarchy)

`finally` block is always executed before exiting the `try` statement. `finally` block is optional but can appear only once after the `catch()` blocks

If there is no `finally` block, then at least one `catch()` block must appear after the `try` statement.

Unhandled Exceptions



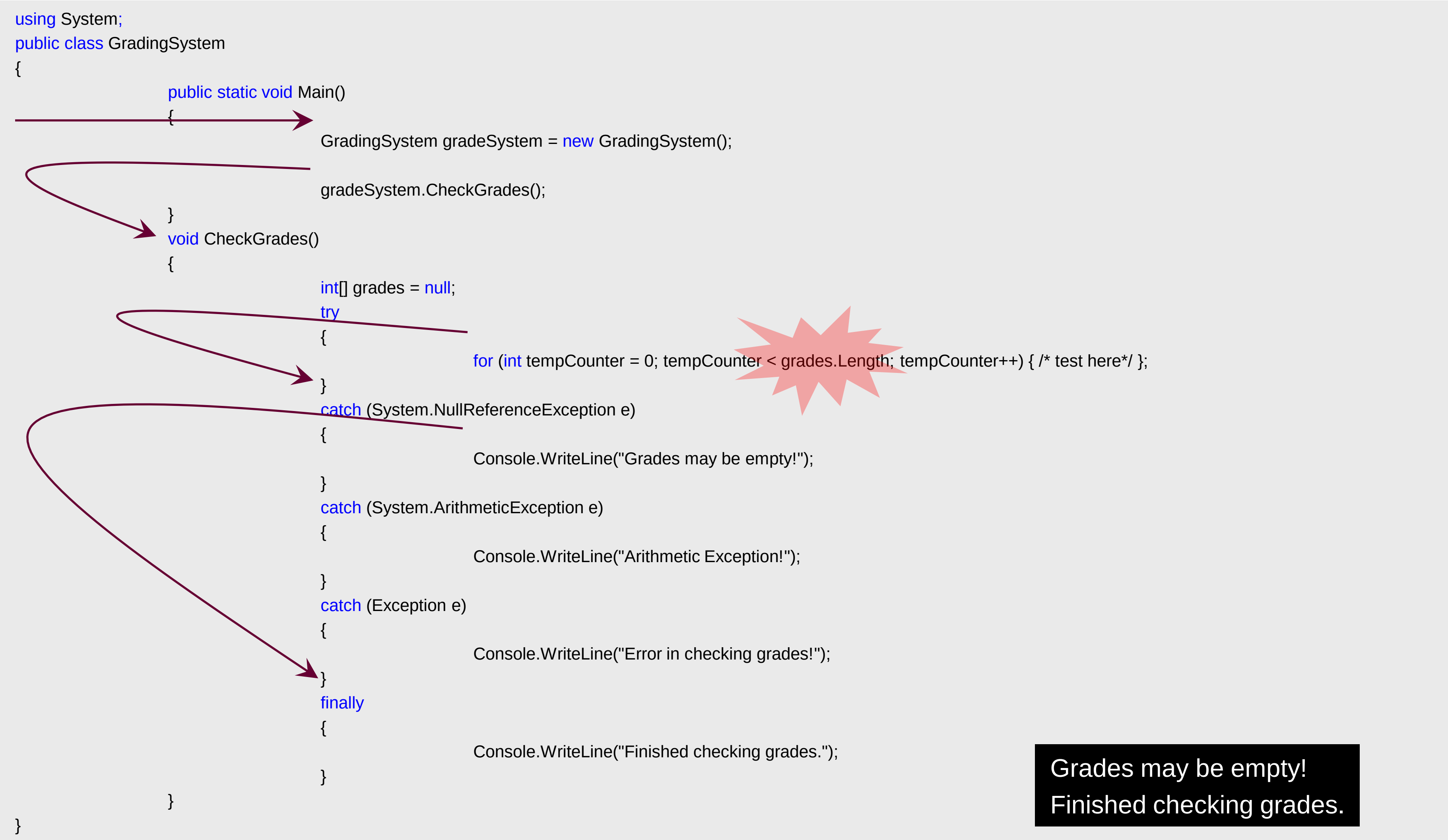
```
public class GradingSystem
{
   public static void Main()
    {
          GradingSystem gradeSystem = new GradingSystem();
          gradeSystem.CheckGrades();
    }
   void CheckGrades()
    {
          int[] grades = null;

          for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++)
          { /* test here*/ };
    }
}
```

```
Unhandled Exception: System.NullReferenceException: Object reference not set to an
instance of an object.
   at TestConsoleApps.GradingSystem.CheckGrades()

   at TestConsoleApps.GradingSystem.Main()
```

Handled Exceptions



```
using System;
public class GradingSystem
{
        public static void Main()
        {
                GradingSystem gradeSystem = new GradingSystem();

                gradeSystem.CheckGrades();
        }
        void CheckGrades()
        {
                int[] grades = null;
                try
                {
                        for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++) { /* test here*/ };
                }
                catch (System.NullReferenceException e)
                {
                        Console.WriteLine("Grades may be empty!");
                }
                catch (System.ArithmeticException e)
                {
                        Console.WriteLine("Arithmetic Exception!");
                }
                catch (Exception e)
                {
                        Console.WriteLine("Error in checking grades!");
                }
                finally
                {
                        Console.WriteLine("Finished checking grades.");
                }
        }
}
```

```
Grades may be empty!
Finished checking grades.
```

Propagating Exceptions

```
using System;
public class GradingSystem {
   public static void Main() {
      GradingSystem gradeSystem = new GradingSystem();
      try {
         gradeSystem.CheckGrades();
       }
       catch (Exception e) {            // must handle the exception thrown
          Console.WriteLine(e.Message);
       }
   }
   void CheckGrades() {
      int[] grades = {81,0,75};
      try {
         for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++) {
            if (grades[tempCounter] <= 0) {
               throw new Exception("Invalid grade!");
            }
         }
      }
      catch (System.NullReferenceException e) {
          Console.WriteLine("Grades may be empty!");
      }
      catch (System.ArithmeticException e) {
         Console.WriteLine("Problem while executing!");
      }
      catch (Exception e) {
         Console.WriteLine("Can't handle error here! Rethrowing...");
         throw new Exception(e.Message);
      }
   }
}
```

```
Can’t handle error here! Rethrowing…
Invalid Grade!
```

Common Exception Classes

- Common exception classes in C# are:
  - *System.ArgumentException*
    - A base class for exceptions that occur when one of the arguments provided to a method is not valid, such as *System.ArgumentNullException* and *System.ArgumentOutOfRangeException*.
  - *System.NullReferenceException*
    - Thrown when a null reference is used in a way that causes the referenced object to be required.
  - *System.DivideByZeroException*
    - Thrown when an attempt to divide an integral value by zero occurs.
  - *System.IndexOutOfRangeException*
    - Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
  - *System.InvalidCastException*
    - Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.

User-Defined Exceptions

- Following is an example of a User-Defined Exception.

```
using System;
[Serializable]
public class EmployeeListNotFoundException : Exception
{
   public EmployeeListNotFoundException()
   { }

   public EmployeeListNotFoundException(string message)
       : base(message)
   { }

   public EmployeeListNotFoundException(string message, Exception inner)
      : base(message, inner)
   { }

   // Full .NET only: to work correctly across application domain and remoting boundaries,
   // you have to define:
   protected EmployeeListNotFoundException(SerializationInfo info, StreamingContext context)
      : base(info, context)
   { }
}
```

C# 6 Exception Filters

```
using static System.Console;

class Program
{
    static void ExceptionFilterDemo()
    {
        bool mustCatch = false;
        try
        {
            :::
            mustCatch = true;
            :::
        }
        catch (ArgumentNullException e) when (!System.Diagnostics.Debugger.IsAttached)
        {
            WriteLine(e.Message);
        }
        catch (Exception e) when (mustCatch)
        {
            WriteLine(e.Message);
        }
    }
}
```

Best Practices for Handling Exceptions (1 of 4)

- The following list contains suggestions on best practices for handling exceptions:
  - Know when to set up a try/catch block
    - For example, you can programmatically check for a condition that is likely to occur without using exception handling.
    - In other situations, using exception handling to catch an error condition is appropriate.
  - Use try/finally blocks around code that can potentially generate an exception and centralize your catch statements in one location
    - In this way, the try statement generates the exception, the finally statement closes or deallocates resources, and the catch statement handles the exception from a central location.

Best Practices for Handling Exceptions (2 of 4)

- The following list contains suggestions on best practices for handling exceptions:
  - Always order exceptions in catch blocks from the most specific to the least specific
    - This technique handles the specific exception before it is passed to a more general catch block.
  - Try to avoid empty catch blocks.
  - End exception class names with the word "Exception“.
  - In C#, use at least the three common constructors when creating your own exception classes.
  - In most cases, use the predefined exceptions types. Define new exception types only for programmatic scenarios.

Best Practices for Handling Exceptions (3 of 4)

- More suggestions on best practices for handling exceptions:
  - Do not derive user-defined exceptions from the ApplicationException base class
    - For most applications, derive custom exceptions from the *Exception* class.
  - Include a localized description string in every exception
    - When the user sees an error message, it is derived from the description string of the exception that was thrown, rather than from the exception class.
  - Use grammatically correct error messages, including ending punctuation
    - Each sentence in a description string of an exception should end in a period.

Best Practices for Handling Exceptions (4 of 4)

- More suggestions on best practices for handling exceptions:
  - Provide exception properties for programmatic access
    - Include extra information in an exception (in addition to the description string) only when there is a programmatic scenario where the additional information is useful.
  - Return null for extremely common error cases
  - Design classes so that an exception is never thrown in normal use
  - The stack trace begins at the statement where the exception is thrown and ends at the catch statement that catches the exception
    - Be aware of this fact when deciding where to place a throw statement
  - Use exception builder methods
    - It is common for a class to throw the same exception from different places in its implementation
    - To avoid excessive code, use helper methods that create the exception and return it

Key Points

- When your application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated.
- Once an exception occurs, the flow of control immediately jumps to an associated exception handler, if one is present.
- If no exception handler for a given exception is present, the program stops executing with an error message.
- Actions that may result in an exception are executed with the try keyword.
- An exception handler is a block of code that is executed when an exception occurs. In C#, the catch keyword is used to define an exception handler.
- Exceptions can be explicitly generated by a program using the throw keyword.
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error.
- Code in a finally block is executed even if an exception is thrown, thus allowing a program to release resources.

Questions and Comments

- What questions or comments do you have?